Starvation-free mutual exclusion with semaphores

Hesselink, Wim H.; IJbema, Mark

Published in:
Formal Aspects of Computing

DOI:
10.1007/s00165-011-0219-y

Publication date:
2013

DOI 10.1007/s00165-011-0219-y

Formal Aspects of Computing (2013) 25: 947–969

Formal Aspects of Computing

Starvation-free mutual exclusion with semaphores

Wim H. Hesselink and Mark IJbema

Department of Computing Science, University of Groningen, P.O. Box 407, 9700 AK Groningen, The Netherlands.
E-mail: w.h.hesselink@rug.nl; markijbema@gmail.com

Abstract. The standard implementation of mutual exclusion by means of a semaphore allows starvation of processes. Between 1979 and 1986, three algorithms were proposed that preclude starvation. These algorithms use a special kind of semaphore. We model this so-called buffered semaphore rigorously and provide mechanized proofs of the algorithms. We prove that the algorithms are three implementations of one abstract algorithm in which every competing process is overtaken not more than once by any other process. We also consider a so-called polite semaphore, which is weaker than the buffered one and is strong enough for one of the three algorithms. Refinement techniques are used to compare the algorithms and the semaphores.

Keywords: semaphore, binary semaphore, mutual exclusion, refinement, starvation freedom

1. Introduction

The semaphore was introduced by E.W. Dijkstra around 1965 to allow synchronization and communication between co-operating sequential processes [Dij68a], initially on the THE operating system [Dij68b]. The first main application was to ensure that such processes could access a common resource under mutual exclusion.

In recent times, the semaphore has lost large parts of its application area, on the one hand to monitors as provided (e.g.) by Java [Lea00] with its synchronized methods and blocks, and on the other hand to other locking mechanisms like mutexes and condition variables [But97]. Monitors form a more structured synchronization mechanism than semaphores and mutexes. The main difference between a semaphore and a mutex is that a semaphore can be released by any process, while a mutex can only be released by the process that has acquired the mutex (e.g. [And00, p. 247]). Semaphores are thus more powerful than mutexes. A second difference is that a general semaphore counts the number of unused release actions, while a mutex forgets them or throws an exception. We do not use this second difference and restrict ourselves mainly to binary semaphores.

Between 1979 and 1986, three papers [Mor79, MB85, Udd86] appeared with algorithms to improve the fairness properties of semaphores when used to establish mutual exclusion. All three papers offer proofs which, in our view, are not detailed enough to be convincing. In particular, they are not explicit about the modelling of semaphores and the atomic steps of the algorithms.

Correspondence and offprint requests to: W. H. Hesselink, E-mail: w.h.hesselink@rug.nl

In this paper, we present these algorithms as implementations of one abstract algorithm, together with proofs of the abstract algorithm and of the refinement relations for two of them. In addition to the ordinary semaphore, which we call the plain one, we discuss two stronger versions: the buffered semaphore and the polite semaphore. The buffered semaphore is the one intended in the three papers. The polite one is weaker. It was suggested by the phrasing in [Udd86], but is applicable only in the algorithm of [MB85].

1.1. Overview

In Sect. 2, we introduce mutual exclusion, and plain and buffered semaphores. Section 3 presents our formal concurrency model and simulation methods to relate algorithms of different levels of abstraction. Section 4 contains an abstract algorithm for fair mutual exclusion that generalizes all three algorithms. Morris' algorithm [Mor79] is developed in Sect. 5, with a proof of mutual exclusion and a construction of a refinement to the abstract algorithm. We discuss Udding's algorithm [Udd86] briefly in Sect. 6. It is so similar to Morris' algorithm that a separate proof is not illuminating. In Sect. 7, we introduce the polite semaphore, present the algorithm of Martin and Burch [MB85], and prove the correctness of this algorithm when using a polite semaphore. Section 8 contains the proof that the buffered semaphore is stronger than the polite one. The verifications have been carried out with the proof assistant PVS [OSRSC01]. The proof scripts are available at [Hes11].

1.2. Contributions

1. Distinction and formalization of plain, buffered, and polite semaphores.
2. Formal verification of the algorithms of Morris, Udding, and Martin–Burch, with proof that the third one only needs a polite semaphore.
3. Unification of the algorithms by refinement towards one abstract algorithm.

2. Mutual exclusion and semaphores

We introduce the mutual exclusion problem in Sect. 2.1. Its progress requirements are described in Sect. 2.2. Atomicity of commands is discussed in Sect. 2.3. Section 2.4 introduces plain semaphores, and shows how they can be used for mutual exclusion. In Sect. 2.5 we discuss stronger requirements for semaphores. This is related to the distinction between weak and strong fairness in Sect. 2.6. The buffered semaphore is introduced in Sect. 2.7.

In shared variable concurrency, the shared state space is spanned by the shared variables, and the private state space of any process q is spanned by the private variables of q. We use `typewriter` font for shared variables and *slanted* font for private variables. When we reason about a program, the value of a private variable v of process q is denoted by $v.q$. The location of process q in its code is given by the private variable $pc.q$ (program counter of q). Formally, the locations are the labels of the atomic statements of the program. If ℓ is a location, we write q **at** ℓ to mean $pc.q = \ell$. If L is a set of locations, we write q **in** L to mean $pc.q \in L$.

2.1. Mutual exclusion

The mutual exclusion problem was first formulated in [Dij65]. It is traditionally modelled in the following way. There are several concurrent processes that, from time to time, need to execute a critical section *CS* and that at other times execute a noncritical section *NCS*. We therefore assume that the processes never terminate, and that they execute the following code:

process (p) =
  loop
    NCS ; *Entry* ; *CS* ; *Exit*
  endloop.

The program fragments *NCS*, *Entry*, *CS*, *Exit* may depend on p. We prefer to keep this implicit, as e.g. in [LH91]. We use *NCS* and *CS* also to denote the sets of their locations.

Mutual exclusion means that never two different processes are in *CS* at the same time. This is expressed by requiring for all processes q and r:

MX: q **in** *CS* $\wedge$ r **in** *CS* $\Rightarrow$ $q = r$.

A process is defined to be *participating* when it is not in *NCS*. The mutual exclusion problem is to implement *Entry* and *Exit* in such a way that *MX* holds, and that every participating process has a "fair chance" to (execute *CS* and) reach *NCS* again eventually.

Indeed, execution of *NCS* need not terminate, because the processes need not be interested in the critical section. Of course, when a process is in *CS*, other participating processes may have to wait in or before *Entry*. Therefore, execution of *CS* is supposed always to terminate.

2.2. Progress for mutual exclusion

Following [LH91], a mutual exclusion algorithm is called *deadlock-free* if, whenever there are participating processes, eventually some participating process exits. It is called *starvation-free* [Dij77] when every participating process eventually exits.

In order to formulate more precise conditions, we assume that *Entry* is a sequential composition of a *doorway* [Lam74] that can be passed without waiting, and a waiting section. We define a process to be *competing* when it is participating and has passed the doorway. Indeed, in many algorithms, the doorway is used for registration of participating processes.

A process is said to have *bounded overtaking* (with bound $k > 0$) if, for any two processes q and r, during any period in which q is competing, process r does not exit more than k times. It has the *first-come-first-served* property (FCFS) [Lam74] if it is not possible that, while some process q is competing and has not yet reached *CS*, some other process starts with *Entry* and reaches *CS* before q does.

Clearly, every starvation-free algorithm is deadlock-free. It is also clear that FCFS implies bounded overtaking with bound 1. Finally, a deadlock-free algorithm with bounded overtaking is starvation-free, provided the number of processes is finite.

Deadlock-freedom as defined above is a progress property. There is a closely related safety property that often goes by the same name. We say that the system is free from *immediate deadlock* if, whenever there are participating processes, at least one participating process can do a computation step. This is a safety property that is necessary but not sufficient for deadlock-freedom.

2.3. Atomicity

In concurrency, the atomic commands of the processes are interleaved in arbitrary way. It is therefore important to specify the grain of atomicity of the commands.

According to the *principle of single critical reference*, e.g., [OG76, (3.1)], [AdBO09, p. 273], an atomic command reads or writes at most one shared variable (not both), unless it is specifically provided as a call to the operating system (e.g., a semaphore operation). Actions on private variables can be added to atomic commands because they never lead to interference.

This principle sets the default. In the design of algorithms and in the specification of such operating system calls, however, we often need to deviate from the principle.

A command S is enclosed in angled brackets $\langle S \rangle$ to express that it is to be executed atomically, i.e., without interference by other processes. The await statement $\langle$**await** $B \rightarrow S\rangle$ is used to specify that execution needs to wait for validity of condition B as a precondition for the action S. The atomicity brackets express that the enclosed command, i.e., the evaluation of B, if *true* followed by S, is to be executed atomically. See [And00, p. 54]. We use these constructs below to specify semaphores. In concrete algorithms, we often label the atomic commands. Then the labels determine the atomicity, and we do not need the atomicity brackets $\langle\ \rangle$.

2.4. Using plain semaphores

Recall that a *semaphore* [Dij68a, And00, AdBO09, HS08, MvdS89] is a shared integer variable $\mathtt{s}$, with some initial value, which can only be accessed by operations $\mathbf{P}(\mathtt{s})$ and $\mathbf{V}(\mathtt{s})$, given by

(0) $\quad \mathbf{P}(\mathtt{s}) \;=\; \langle\ \mathbf{await}\ \mathtt{s} > 0 \;\rightarrow\; \mathtt{s} := \mathtt{s} - 1\ \rangle,$
$\quad\quad\ \mathbf{V}(\mathtt{s}) \;=\; \langle\ \mathtt{s} := \mathtt{s} + 1\ \rangle.$

One can use a semaphore with initial value $\mathtt{s} = 1$ to implement mutual exclusion by:

(1)
process $(p) =$
  loop
    $NCS\ ;\ \mathbf{P}(\mathtt{s})\ ;\ CS\ ;\ \mathbf{V}(\mathtt{s})$
  endloop.

This satisfies mutual exclusion because a system of such processes preserves the invariant that the number of processes in CS equals $1 - \mathtt{s}$:

(sem) $\quad \#\{q \mid q\ \mathbf{in}\ CS\} = 1 - \mathtt{s}.$

Indeed, initially, we have $\mathtt{s} = 1$ and there are no processes in CS. Whenever a process enters CS, the value of $\mathtt{s}$ is decremented with 1. It is incremented with 1 whenever some process exits from CS. Formula (sem) implies mutual exclusion because the semaphore preserves the invariant $\mathtt{s} \geq 0$. Formula (sem) also implies that the semaphore is binary, i.e., satisfies the invariant $\mathtt{s} \leq 1$. In this paper, most semaphores are binary.

More generally, a system of semaphores $\mathtt{s}_1, \ldots, \mathtt{s}_n$ forms a so-called *split binary semaphore* [Hoa74] if the system preserves the invariant

(sbs) $\quad \mathtt{s}_1 + \cdots + \mathtt{s}_n + \#\{q \mid q\ \mathbf{in}\ PV\} = 1,$

where PV is a set of locations. The invariant implies that a process can only enter PV by executing $\mathbf{P}(\mathtt{s}_i)$ for some i, and only exit PV by executing $\mathbf{V}(\mathtt{s}_j)$ for some j. The choice of the indices i and j makes it possible to implement general condition synchronization [And00, Section 4.4].

2.5. Stronger semaphores

System (1) of the previous section is deadlock-free, but not starvation-free. It does not guarantee that a process interested in reaching CS will ever reach it, because it may be that whenever the process tests the condition $\mathtt{s} > 0$ there is another process that has entered CS.

A semaphore is called *strong* when it guarantees that algorithm (1) is starvation-free. In [Dij77], Dijkstra conjectured that starvation-free mutual exclusion for an unbounded number of processes cannot be implemented using a fixed number of weak semaphores, but he did not explicitly define *weak* semaphores.

Dijkstra had proposed the semaphore for a uniprocessor system, and he expected implicitly that the operating system would attach a buffer or queue to every semaphore $\mathtt{s}$ in such a way that with every $\mathbf{V}(\mathtt{s})$ operation one of the waiting processes can be allowed to pass the semaphore. We call such a semaphore a *buffered* semaphore. We call a semaphore *plain* when it only satisfies the formulas of (0).

In Dijkstra's conjecture, we therefore interpret the *weak* semaphores as buffered ones. This conjecture was then refuted by Morris in [Mor79]. Similar algorithms appeared in, e.g., [MB85, Udd86]. These three papers present starvation-free mutual exclusion algorithms that each need one buffered semaphore and one or two plain ones. These algorithms satisfy bounded overtaking with bound 2.

2.6. An aside on fairness

Parallel to the distinction between weak and strong semaphores, in the 1980s, also the distinction emerged between weak and strong fairness, see e.g. [LPS81, Fra86, And00].

Weak fairness means that, when a process from some time onward is continuously enabled to do a step, it will do the step. *Strong fairness* means that, when a process from some time onward is infinitely often enabled to do a step, it will do the step.

Weak fairness is a natural assumption, which is usually taken for granted, or postulated as e.g. in [CM88]. Strong fairness is regarded as unrealistic, because it requires a scheduler with information about enabledness of processes [Hes06, Section 4]. Strong fairness would imply that system (1) with plain semaphores is starvation-free. It would thus trivially refute Dijkstra's conjecture.

2.7. Buffered semaphores

The refutation papers [Mor79, MB85, Udd86] did not formally model the weak semaphores used. Morris [Mor79] assumes the following property for the weak semaphore: "If any process is waiting to complete a **P**(s) operation, and another process performs a **V**(s), the semaphore is not incremented and one of the waiting processes completes its **P**(s)." This property is used by Tanenbaum [Tan08] as the definition of the semaphore.

Martin and Burch [MB85] give the following axioms for the weak semaphore:

$$\begin{array}{ll} \mathtt{s} \geq 0 & \wedge \quad \mathbf{cP}\mathtt{s} + \mathtt{s} = \mathbf{cV}\mathtt{s} + \mathtt{s0}, \\ \mathbf{qP}\mathtt{s} = 0 & \vee \quad \mathtt{s} = 0 \end{array} \tag{2}$$

where s0 is the initial value of s, **cP**s and **cV**s are the number of completed **P** and **V** operations on s, and **qP**s is the number of currently suspended **P** operations. This is an incomplete axiomatization because it does not make explicit which processes are suspended, or what it means to be suspended. The difficulty is to distinguish between processes that are suspended, and processes that are ready to pass the semaphore when scheduled. Even the plain semaphore satisfies the axioms (2) if we take **qP**s to be identically 0.

The phrase "waiting to complete a **P** operation" means that the **P** operation consists of at least two atomic steps. As we do not want to let two processes do a synchronous atomic step, we propose the following formalization.

We define a *buffered* semaphore s as a shared variable that consists of a natural number s and a set buf of waiting processes that is initially empty. Its operations **V** and **P** are given by

V(s) : ⟨ **if** *empty?*(buf) **then** s := s + 1
else *remove some q from* buf **endif** ⟩.

P(s) : ⟨ **if** s > 0 **then** s := s − 1 ; **goto** ℓ
else buf := buf ∪ {*self*} **endif** ⟩ ;
⟨ **await** *self* ∉ buf ⟩ ;
ℓ : ...

Here, *self* is the (process identifier of the) acting process. The **P** operation is thus split in two atomic steps. If s = 0, the acting process enters the buffer, and waits until it is released from the buffer by a **V**(s) operation of another process. Otherwise, it decrements s. In any case, the first step of **P** is always enabled. It can therefore serve as the doorway when the semaphore is used for mutual exclusion as in (1). System (1) with such a semaphore, however, is not starvation-free when there are more than two processes.

3. A formal model for concurrency

In order to prove or refute assertions about concurrent algorithms, we need a formal model. This model also enables us to perform mechanical verification by means of a theorem prover. We use the proof assistant PVS [OSRSC01] for this purpose.

In fact, in our view, the only way to be certain about aspects of the behaviour of a system of concurrent processes is to analyse which properties *are not made false by any step of the system*. These properties are called the invariants of the system. As the system at any moment can do many different steps, the proof of invariance usually requires many case distinctions. Humans are not very good at this, because we tend to ignore seemingly innocent cases. Formalization with a theorem prover is indispensable.

In Sect. 3.1, we introduce transition systems to model concurrent algorithms. In order to compare related transition systems, we introduce refinement functions in Sect. 3.2, and strict simulations in Sect. 3.3.

3.1. Transition systems

A *transition system* is a triple (X, A, N) where X is a set, called the *state space*, A is a subset of X (the initial states), and N is a binary relation on X, called the *next state* relation. Relation N is required to be reflexive, i.e. the identity relation 1_X must be a subset of N, to allow stuttering. For simplicity, we do not include liveness

properties as used e.g. in [AL91, Hes08] in this formal model. If $K = (X, A, N)$, we write $X = X_K$, $A = A_K$, $N = N_K$.

An *execution* of transition system K is an infinite sequence of states $x_n \in X_K$ with $x_0 \in A_K$ and $(x_n, x_{n+1}) \in N_K$ for all n. A state $x \in X_K$ is said to be *reachable* if it occurs in an execution.

A set $J \subseteq X_K$ is said to be an *invariant* of K if it contains all reachable states. A set J is said to be *inductive* if $A_K \subseteq J$ and $y \in J$ for all $(x, y) \in N_K$ with $x \in J$. It follows that a set J is an invariant if and only if it contains an inductive set.

When dealing with concurrency, the next state relation is of the form $N = 1_X \cup \bigcup_p N_p$ where p ranges over the set of processes (threads). Relation N_p is given by the programming code of process p. The pairs in N_p are called *atomic steps* of process p. Usually, the state space X is a Cartesian product of a shared state space with private state spaces of the processes, where a step of process p can only inspect and modify the components in the shared state space and in the private state space of process p.

3.2. Refinement functions

We use the concept of simulation to formalize implementation relations. The idea is that a transition system K simulates a transition system L if every execution of K mimics some execution of L. In that case, K may be regarded as a concrete system that implements the abstract system L. Note that K need not be able to mimic every execution of L. For instance, a stack implements a multiset, even though not every execution of the multiset can be mimicked by the stack.

The word *mimic* has several different formalizations. They are presented here in increasing generality. The easiest way to prove that one transition system simulates the behaviour of another one, is by means of a function.

A function between the state spaces of K and L, say $f : X_K \to X_L$, is defined to be a *refinement mapping* from K to L [AL91] if it satisfies $f(x) \in A_L$ for every $x \in A_K$, and $(f(x), f(y)) \in N_L$ for every $(x, y) \in N_K$.

Slightly more general, we define a function f to be a *refinement function* from K to L if it is a function defined on an invariant J of K and satisfies $f(x) \in A_L$ for every $x \in A_K$, and $(f(x), f(y)) \in N_L$ for every $(x, y) \in N_K$ with $x, y \in J$. The difference between refinement mappings and refinement functions is a mere formality. Informally, one is not interested in the unreachable states and one tends to ignore them. This is formally justified by using a refinement function instead of a refinement mapping.

3.3. Strict simulations

Functions are often too rigid. Therefore, in 1971, Milner [Mil71] introduced simulation relations, which later were called forward simulations [LV95] or downward simulations [HHS86]. We prefer the first alternative.

A binary relation between the state spaces of K and L, say $F \subseteq X_K \times X_L$, is called a *forward simulation* from K to L if:

(a) for every $x \in A_K$ there is some $u \in A_L$ with $(x, u) \in F$,
(b) whenever $(x, y) \in N_K$ and $(x, u) \in F$ there is some $v \in X_L$ with $(y, v) \in F$ and $(u, v) \in N_L$.

We define a binary relation $F \subseteq X_K \times X_L$ to be a *strict simulation* from K to L [Hes08] if, for every execution $(x_n \mid n \in \mathbb{N})$ of K, there is an execution $(y_n \mid n \in \mathbb{N})$ of L with $(x_n, y_n) \in F$ for all n.

Every forward simulation is a strict simulation. Indeed, given an execution $(x_n \mid n \in \mathbb{N})$ of K, one constructs a matching execution $(u_n \mid n \in \mathbb{N})$ of L by first choosing $u_0 \in A_L$ with $(x_0, u_0) \in F$. When u_n has been chosen with $(x_n, u_n) \in F$, one chooses u_{n+1} such that $(x_{n+1}, u_{n+1}) \in F$ and $(u_n, u_{n+1}) \in N_L$.

If $f : X_K \to X_L$ is a refinement function from K to L, the graph $\{(x, f(x)) \mid x \in X_K\}$ is a forward simulation from K to L, and hence a strict simulation.

A relational composition of strict simulations is a strict simulation. Therefore, a composition of a forward simulation and a refinement function is a strict simulation.

In the Sects. 5.3 and 7.5, we use forward simulations and refinement functions to relate the algorithms of Morris and Martin–Burch to the abstract algorithm of the next section.

4. An abstract algorithm

The algorithms of [Mor79, MB85, Udd86] are three different implementations of one abstract algorithm that establishes mutual exclusion with bounded overtaking. We present this abstract algorithm first, because it is easier to explain, and it nicely motivates Morris' algorithm.

The abstract algorithm can be explained by the elevator analogue. While there are interested processes, they enter the elevator at the first floor. When there are no processes arriving anymore, the elevator goes to the second floor and lets its occupants into *CS*, one by one. When the elevator is empty, it goes down again. When the elevator is not at the first floor, arriving processes have to wait. After *CS*, the processes go down by the stairs.

The abstract program uses the shared variables:

var se, sm : $\mathbb{N}$,
var ne, nm : $\mathbb{Z}$.

The variables se and sm serve as semaphores that model the doors of the elevator at the first and second floor, respectively. The variables ne and nm count the number of processes waiting for the elevator at the first floor and within the elevator, respectively. The initial conditions are

$$\texttt{ne} = \texttt{nm} = \texttt{sm} = 0 \ \wedge \ \texttt{se} = 1.$$

(3)
```
process (p) =
    loop
 9      NCS ;
10      ne++ ;
11      await se > 0 ; nm++ ; ne-- ;
        if ne = 0 then sm++ ;  se-- endif ;
12      await sm > 0 ; sm-- ; nm-- ;
13      CS ;
14      if nm > 0 then sm++ else se++ endif
    endloop.
```

The numbered commands are regarded as atomic. Mutual exclusion *MX* is the requirement that there is at most one process at 13. So, even though *CS* is treated as atomic, *MX* has the intended meaning.

We call algorithm (3) an abstract algorithm because it violates the principle of single critical reference of Sect. 2.3 in the commands 10, 11, 12, 14 (in 10, variable ne is both read and written).

The declarations of se, sm as naturals, and ne, nm as integers are subtleties to simplify the mechanical proof. Firstly, the atomic steps of (3) preserve the invariants that se and sm are natural, but these invariants become superfluous by their declaration as naturals. On the other hand, viewed in isolation, the atomic steps can make ne and nm negative. Therefore, in PVS, declaration of them as naturals would create unprovable type correctness conditions for the atomic steps.

The variable ne holds the number of processes at line 11, and nm holds the number of processes at line 12, according to the obvious invariants:

Rq0: $\#\{q \mid q \textbf{ at } 11\} = \texttt{ne}$,
Rq1: $\#\{q \mid q \textbf{ at } 12\} = \texttt{nm}$.

We give invariants names of the form *Xqd* where *X* is an uppercase letter, *q* is fixed, and *d* is a digit, to make it easier to preserve the correspondence between the paper and the PVS proof while they evolve.

The variables se and sm together serve as a split binary semaphore, in the sense that we have the easy invariant:

Rq2: $\texttt{se} + \texttt{sm} + \#\{q \mid q \textbf{ in } \{13, 14\}\} = 1$.

This invariant implies mutual exclusion because se and sm are natural numbers so that the third summand of the lefthand side of *Rq2* is ≤ 1.

We turn to the question of bounded overtaking. In the elevator analogue, consider the following scenario. Process q_0 arrives when the elevator is up. Process q_1 exits the elevator at *CS*, goes down again, and enters the elevator together with q_0. When the elevator is again at the second floor, q_1 enters *CS* first, and exits. Then q_0 enters *CS* and exits. During this competing period of q_0, process q_1 has exited twice. It is easy to

see that this is the worst-case scenario. This shows informally that the algorithm has bounded overtaking with bound 2.

For the formal treatment, recall that a process is called competing when it has passed the doorway, and that the doorway is the part of *Entry* that can be passed without waiting. In the present case, the doorway is line 10. A process is therefore competing when it is in $\{11 \dots 14\}$. In order to prove bounded overtaking, for every pair of processes q, r, we construct a variant function $vf(q, r) : \mathbb{N}$ that never increases while process q is competing, and that decreases whenever process r executes line 14 while process q remains competing. By construction, we shall have $vf(q, r) \leq 2$. So, this implies bounded overtaking with bound 2.

We define vf by:

$$vf(q, r) = (\mathtt{se} > 0 \,?\, 1 : 1 - phase(q) + phase(r)),$$
where $phase(p) = (pc.p \leq 11 \,?\, 0 : 1)$.

Indeed, it is now easy to see that $0 \leq vf(q, r) \leq 2$.

Theorem 1 *For every pair q, r, we have that in any step of the algorithm in which process q is competing and remains competing, $vf(q, r)$ does not increase.*

Proof Assume that some process p does a step that increases $vf(q, r)$ while q remains competing. Note that p can be equal to q or r, or differ from both. We distinguish three cases.

If p does not modify se, then $\mathtt{se} = 0$ and p modifies the phases of q or r. It cannot increment $phase(r)$ because of $\mathtt{se} = 0$. It cannot decrement $phase(q)$ because q remains competing. Therefore p modifies se.

Assume that p increments se. Then p is at 14 and $\mathtt{nm} = 0$. Moreover $vf(q, r) = 0$, because p increments it and it becomes 1. Therefore $phase(q) = 1$ and q is in $\{12 \dots 14\}$. Because p is at 14 and q remains competing, *Rq2* implies that q is at 12, contradicting *Rq1*.

Assume that p decrements se. Then p is at 11, $\mathtt{se} = 1$, $\mathtt{ne} = 1$, $phase(q) = 0$, and $phase(r) = 1$. Because q is competing, it is at 11. Therefore *Rq0* implies $p = q$, and the step establishes $phase(q) = 1$, and $vf(q, r)$ does not increase. □

Theorem 2 *For every pair q, r, if process q is competing and process $r \neq q$ performs step 14, $vf(q, r)$ decreases.*

Proof Because process r performs step 14, we have $\mathtt{se} = 0$ by *Rq2* and the step decreases $phase(r)$. If se remains 0 in the step, $vf(q, r)$ decreases. If se becomes 1, then $\mathtt{nm} = 0$, so that, by *Rq1* and *Rq2*, process q is at 11 and $phase(q) = 0$. We therefore have $vf(q, r) = 2$ in the precondition, and $vf(q, r)$ decreases because se becomes 1. □

These results together imply for every q and r that, while process q is competing, process r can execute *CS* and exit not more than two times.

This result is only useful, however, if the algorithm is deadlock-free. For the proof of this, we need two more invariants:

Rq3: $\mathtt{sm} > 0 \;\Rightarrow\; \mathtt{nm} > 0,$
Rq4: $\mathtt{se} > 0 \;\wedge\; \mathtt{nm} > 0 \;\Rightarrow\; \mathtt{ne} > 0.$

Preservation of *Rq3* follows at 11 from *Rq1*, and at 12 from *Rq2*. Preservation of *Rq4* follows at 11 from *Rq0* and *Rq2*.

We first turn to the question of absence of immediate deadlock. Recall that this means that, when there are participating processes, there is a participating process that can do a step (i.e. is enabled).

Theorem 3 *System* (3) *has absence of immediate deadlock.*

Proof Assume that all participating processes are disabled. We then have:

$$\forall q :\; q \textbf{ at } 9 \;\vee\; (q \textbf{ at } 11 \;\wedge\; \mathtt{se} = 0) \;\vee\; (q \textbf{ at } 12 \;\wedge\; \mathtt{sm} = 0).$$

Using *Rq0* and *Rq1*, it follows that ne + nm is the number of participating processes. Using *Rq2*, it follows that se + sm = 1. Because there are no enabled processes at 11 or 12, the invariants *Rq0* and *Rq1* imply

$$\mathtt{se} > 0 \;\Rightarrow\; \mathtt{ne} = 0,$$
$$\mathtt{sm} > 0 \;\Rightarrow\; \mathtt{nm} = 0.$$

Therefore *Rq3* implies that $\mathtt{sm} = 0$ and $\mathtt{se} = 1$. It follows that $\mathtt{ne} = 0$ and, by *Rq4*, also that $\mathtt{nm} = 0$. This proves that the number of participating processes is 0. □

Theorem 3 implies deadlock-freedom because system (3) has no inner loops. Strictly speaking, this requires that there are only finitely many processes. Indeed, if there were infinitely many processes, weakly fair executions would exist in which infinitely many processes execute steps 9, 10, and 11, and the variable ne grows to infinity.

If there are only finitely many processes, one can prove deadlock-freedom in the following way. Give every process a private variable *cnt*, which is initially 0, and which is incremented with 1 whenever the process executes step 14. Define

$$pvf(q) = 6 \cdot cnt.q + pc.q - 9.$$

Every step of process q increments $pvf(q)$. The total sum $Pvf = \sum_q pvf(q)$ is well-defined, and increases with every step of the system. Whenever there are participating processes, Theorem 3 together with weak fairness implies that one of these will do a step and increment Pvf. Therefore, if in some execution Pvf remains bounded, all processes end in *NCS*. Otherwise, Pvf tends to infinity. Because the number of processes is finite, it then follows that there is a process q for which $pvf(q)$ tends to infinity. This process executes *CS* infinitely often. It is not difficult to formalize this with PVS, although we have not done so yet.

5. Morris' algorithm

In this section we present and prove the algorithm of [Mor79]. It is developed in Sect. 5.1 as an informal refinement of the abstract algorithm presented above. In Sect. 5.2 we present the corresponding transition system, and prove mutual exclusion and absence of immediate deadlock. Section 5.3 contains the construction of a simulation towards the abstract algorithm, and shows how this implies bounded overtaking.

5.1. Development of Morris' algorithm

In order to comply with the principle of single critical reference, the composite atomic commands of the abstract algorithm (3) of Sect. 4 have to be split. We first replace the integer shared variables se and sm by plain semaphores. This leads to the following program.

(4)
process (p) =
  loop
    NCS ; ⟨ ne++ ⟩ ;
    P(se) ; nm++ ; ⟨ ne-- ⟩ ;
    if ne $= 0$ **then** **V**(sm) **else** **V**(se) **endif** ;
    P(sm) ; nm-- ; *CS* ;
    if nm > 0 **then** **V**(sm) **else** **V**(se) **endif**
  endloop.

In comparison with (3), the first **else** branch may come as a surprise. Here, se must be incremented because it was decremented by **P**(se), while se can remain constant in this branch in (3).

Algorithm (4) is correct. We have proved this with PVS. It is not entirely obvious, because the shared variable ne can be modified between the decrementation ⟨ne--⟩ and the test $\mathtt{ne} = 0$. We do not describe the proof because it is just a simpler version of the proof of Morris' algorithm that is given below.

The atomic incrementations and decrementations are not allowed by the principle of single critical reference, because they require both reading and writing. The algorithm is useful, however, if one has an atomic counter ne and two plain semaphores. Unfortunately, every implementation of an atomic counter that we know of needs strong fairness.

Morris' algorithm [Mor79] is given in (5). It removes the need for an atomic counter ne by guarding every access to ne by a buffered semaphore sb. Initially, $\mathtt{sb} = \mathtt{se} = 1$ and $\mathtt{sm} = \mathtt{ne} = \mathtt{nm} = 0$.

Syntactically, the program still does not satisfy the principle of single critical reference, but it is acceptable when we regard each incrementation and decrementation as a sequential composition of reading and writing. This is made explicit in the transition system (6) developed below.

(5)
```
process (p) =
    loop
        NCS ;
        P(sb) ; ne++ ; V(sb) ;
        P(se) ; nm++ ; P(sb) ; ne-- ;
        if ne > 0 then V(sb) ;  V(se)
        else V(sb) ;  V(sm) endif ;
        P(sm) ; nm-- ; CS ;
        if nm > 0 then V(sm) else V(se) endif
    endloop.
```

5.2. A transition system for Morris' algorithm

In order to prove the correctness, we model the buffered semaphore sb as described in Sect. 2.7, with a buffer buf. The plain semaphores se and sm are modelled as described in Sect. 2.4.

This algorithm uses the shared variables

$$\textbf{var}\ \mathtt{sb}, \mathtt{se}, \mathtt{sm} : \mathbb{N} ;$$
$$\textbf{var}\ \mathtt{ne}, \mathtt{nm} : \mathbb{Z} ;$$
$$\textbf{var}\ \mathtt{buf} : \mathbb{P}[\mathit{Process}].$$

The initial conditions are

$$\mathtt{ne} = \mathtt{nm} = \mathtt{sm} = 0 \ \wedge\ \mathtt{se} = \mathtt{sb} = 1 \ \wedge\ \mathtt{buf} = \emptyset.$$

As announced, the updates ne++, etc., of the shared variables ne and nm are not supposed to be executed atomically, but are modelled by means of private variables *tmp*, which play the role of registers. In this way, program (5) is transformed into (6).

Note that the code preserves the types of the semaphores se, sm, sb : $\mathbb{N}$. In the lines 23 and 28, we have removed the inspection of the shared variables ne and nm. This is possible because the private variable *tmp* holds the latest value of the relevant shared variable. In this way, system (6) is a minor optimization of (5).

The plain semaphores se and sm form a split binary semaphore because of the invariant:

Mq0: $\quad \mathtt{se} + \mathtt{sm} + \#\{q \mid q \textbf{ in } \{16 \ldots 23,\ 25 \ldots 28\}\} = 1.$

Indeed, initially, the lefthand side of *Mq0* equals 1 because se = 1 and sm = 0. A process enters the set of processes if and only if it executes 15 or 24 and thus decrements se + sm with 1. A process leaves the set if and only if it executes 23 or 28 and thus increments se + sm with 1. This proves that *Mq0* is an invariant.

It follows from *Mq0* that

Mq0d: $\quad q, r \textbf{ in } \{16 \ldots 23,\ 25 \ldots 28\} \ \Rightarrow\ q = r.$

In particular, this means mutual exclusion for *CS*. Note that, in our invariants, we implicitly universally quantify over all free variables (here q and r).

The buffered semaphore sb has the invariants:

Mq1: $\quad \#\{q \mid q \notin \mathtt{buf} \ \wedge\ q \textbf{ in } \{11 \ldots 14,\ 19 \ldots 22\}\} = 1 - \mathtt{sb},$
Mq2: $\quad q \in \mathtt{buf} \ \Rightarrow\ q \textbf{ in } \{11, 19\},$
Mq3: $\quad \mathtt{sb} > 0 \ \Rightarrow\ \mathtt{buf} = \emptyset.$

It follows from *Mq1* and *Mq2* that

Mq1d: $\quad q, r \textbf{ in } \{12 \ldots 14,\ 20 \ldots 22\} \ \Rightarrow\ q = r.$

The variables ne and nm hold the number of processes between 14 and 21, and 18 and 26, respectively:

Nq0: $\quad \#\{q \mid q \textbf{ in } \{14 \ldots 21\}\} = \mathtt{ne},$
Nq1: $\quad \#\{q \mid q \textbf{ in } \{18 \ldots 26\}\} = \mathtt{nm}.$

In order to prove this, we need invariants about the values of *tmp*:

(6)
```
process (p) =
    var tmp : ℤ ;
    loop
 9      NCS ;
10      if sb > 0 then sb-- else
            buf := buf ∪ {p} ;
11          await p ∉ buf endif ;
12      tmp := ne + 1 ;
13      ne := tmp ;
14      if possible remove some q from buf else sb++  endif ;
15      await se > 0 ; se-- ;
16      tmp := nm + 1 ;
17      nm := tmp ;
18      if sb > 0 then sb-- else
            buf := buf ∪ {p} ;
19          await p ∉ buf endif ;
20      tmp := ne − 1 ;
21      ne := tmp ;
22      if possible remove some q from buf else sb++  endif ;
23      if tmp > 0 then se++ else sm++ endif ;
24      await sm > 0 ; sm-- ;
25      tmp := nm − 1 ;
26      nm := tmp ;
27      CS ;
28      if tmp > 0 then sm++ else se++ endif
    endloop.
```

Nq2: $(q \textbf{ at } 13 \;\Rightarrow\; tmp.q = \texttt{ne} + 1) \;\wedge\; (q \textbf{ at } 21 \;\Rightarrow\; tmp.q = \texttt{ne} - 1)$,
Nq3: $(q \textbf{ at } 17 \;\Rightarrow\; tmp.q = \texttt{nm} + 1) \;\wedge\; (q \textbf{ at } 26 \;\Rightarrow\; tmp.q = \texttt{nm} - 1)$.

Preservation of *Nq2* at lines 13 and 21 follows from *Mq1d*; preservation of *Nq3* at 17 and 26 follows from *Mq0d*. While it is easy enough to invent the invariants *Nq2* and *Nq3*, the observation that *Mq1d* and *Mq0d* are needed to prove them was forced upon us by our practice of verifying all invariants with the proof assistant PVS. For the next invariants, we rely on PVS even more strongly. Readers with PVS experience may want to inspect the proof script at [Hes11].

We turn to the question of absence of immediate deadlock. Recall that this means that when there are participating processes, there are participating processes that can do a step. We first note that, in view of *Mq2*, a process q is *disabled* if and only if

(7) $$q \in \texttt{buf} \;\vee\; (q \textbf{ at } 15 \;\wedge\; \texttt{se} = 0) \;\vee\; (q \textbf{ at } 24 \;\wedge\; \texttt{sm} = 0).$$

Two critical invariants for deadlock-freedom are:

Nq4: $\texttt{sm} > 0 \;\Rightarrow\; \exists\, r : r \textbf{ at } 24$,
Nq5: $\texttt{se} > 0 \;\wedge\; q \textbf{ at } 24 \;\Rightarrow\; \exists\, r : r \textbf{ at } 15$.

Preservation of *Nq4* at lines 24 and 28 follows from *Mq0*, *Nq1*, and *Nq6* below. Preservation of *Nq5* at lines 15, 23, and 28 follows from *Mq0*, *Nq1*, and *Nq7* below.

Nq6: $q \textbf{ in } \{27, 28\} \;\Rightarrow\; \texttt{nm} = tmp.q$,
Nq7: $tmp.q > 0 \;\wedge\; q \textbf{ in } \{21 \ldots 23\} \;\Rightarrow\; \exists\, r : r \textbf{ at } 15$.

Preservation of *Nq7* at line 20 uses *Nq0*. Note that we cannot claim that $tmp.q = \texttt{ne}$ when q is at line 23, because line 23 is not guarded by the semaphore sb.

Theorem 4 *System* (6) *has absence of immediate deadlock.*

Proof Assume that all participating processes are disabled. Then *Mq1* with formula (7) implies that $\mathtt{sb} = 1$. Therefore, *Mq3* implies that $\mathtt{buf} = \emptyset$. It follows that all processes are at 9, 15, or 24. Therefore, *Mq0* implies that $\mathtt{se} + \mathtt{sm} = 1$. If $\mathtt{sm} > 0$, *Nq4* implies that there is a process r at 24 which is not disabled. This implies that $\mathtt{sm} = 0$, and hence that $\mathtt{se} > 0$.

Because buf is empty and se is positive, all disabled processes are at 24. If there are disabled processes, *Nq5* therefore implies that there is an enabled process at 15, contradicting the assumption. This proves that there are no participating processes. □

If there are only finitely many processes, Theorem 4 implies deadlock-freedom for algorithm (6) because it has no inner loops. The argument for this is identical to the argument used at the end of Sect. 4 for algorithm (3).

5.3. Simulation

We have developed Morris' algorithm (6) by informal stepwise refinement from the abstract system (3) of Sect. 4. In order to formalize this, we construct a simulation from (6) to (3). Because (3) has bounded overtaking with bound 2, this simulation implies that system (6) has the same property.

The simulation is constructed in two steps: as a composition of a forward simulation that adds some history variables, and a refinement function that forgets the implementation variables, see Sect. 3 (compare [AL91]).

We let the locations of *NCS* and *CS* correspond in the two systems. This implies that the abstract locations 9 and 13 correspond to the concrete locations 9 and 27. It follows that the abstract locations 10 and 14 correspond to the concrete locations 10 and 28.

In order to use simulation to prove bounded overtaking, the doorways of the two systems must correspond. We therefore let the first abstract waiting location 11 correspond to the first concrete waiting location 11.

The concrete location corresponding to the abstract location 12 is critical. The concrete location 24 is too late because the decision taken in line 23 is based on the value of *tmp* obtained in line 20. Even line 20 is too late, because it is possible that some process has executed line 10 but not yet 13, so that the concrete value ne is smaller than the abstract value. We therefore let the abstract location 12 correspond to the concrete location 18 (the alternative 19 seems to be doable).

We thus define the abstract program counter on the concrete state space by

$$\begin{aligned} apc.q = (&pc.q \le 10 \,?\, pc.q \\ &: pc.q \le 18 \,?\, 11 \\ &: pc.q \le 26 \,?\, 12 \;\; : pc.q - 14). \end{aligned}$$

We construct the simulation as a composition of a forward simulation with a refinement function. The forward simulation extends system (6) with four shared history variables that correspond to the shared variables of the abstract program:

var qse, qsm, qne, qnm : $\mathbb{Z}$;

with the initial conditions:

$\mathtt{qne} = \mathtt{qnm} = \mathtt{qsm} = 0 \;\wedge\; \mathtt{qse} = 1.$

We add to the commands at lines 10, 18, 26, and 28 the following modifications of history variables:

```
10a   qne++ ;
18a   qne-- ; qnm++ ;
      if sb > 0 ∧ ne = 1 then qse-- ;  qsm++ endif ;
26a   qnm-- ;  qsm-- ;
28a   if tmp > 0 then qsm++ else qse++ endif.
```

Note that the combined command 18–18a touches two shared variables in one atomic command, viz. sb and ne. This is allowed because ne is not modified and its value is used only for modification of the history variables.

As these extensions of the actions of (6) do not modify the variables of (6) and impose no new conditions, it is clear that there is a forward simulation from the system (6) to this extension.

The refinement function from the extended concrete state space to the abstract state space is defined by promoting the history variables to actual variables and forgetting the implementation variables:

$$
\begin{aligned}
f(x) = (\#\\
&\mathtt{ne} := x.\mathtt{qne},\ \mathtt{nm} := x.\mathtt{qnm},\\
&\mathtt{se} := x.\mathtt{qse},\ \mathtt{sm} := x.\mathtt{qsm},\\
&pc := x.apc\ \#).
\end{aligned}
$$

Here, we use (# and #) as record constructors as in PVS, and x as a variable that ranges over the concrete states.

This definition gives a type conflict because the variables se and sm have the type $\mathbb{N}$, while qse and qsm have the type $\mathbb{Z}$. This conflict is resolved by means of the invariants $\mathtt{qse} \geq 0$ and $\mathtt{qsm} \geq 0$ which follow from *Pq2* and *Qq0* below.

For the proof of refinement we need some further invariants. The history variables qne and qnm satisfy the obvious invariants:

Pq0: $\#\{q \mid q \textbf{ in } \{11 \ldots 18\}\} = \mathtt{qne}$,
Pq1: $\#\{q \mid q \textbf{ in } \{19 \ldots 26\}\} = \mathtt{qnm}$.

Pq0 together with *Mq0*, *Mq1*, *Mq3*, *Nq0* implies

Pq0d: $q \textbf{ at } 18 \ \Rightarrow\ (\mathtt{qne} = 1 \ \equiv\ (\mathtt{sb} = 1 \ \wedge\ \mathtt{ne} = 1))$.

Similarly, *Pq1*, *Nq6*, *Mq0d*, and *Nq1* together imply

Pq1d: $q \textbf{ at } 28 \ \Rightarrow\ \mathtt{qnm} = tmp.q$.

For qse, we need the invariants:

Pq2: $\mathtt{se} \leq \mathtt{qse}$,
Pq3: $q \textbf{ in } \{16 \ldots 19\} \ \Rightarrow\ \mathtt{qse} > 0$,
Pq4: $q \textbf{ at } 20 \ \wedge\ \mathtt{ne} > 1 \ \Rightarrow\ \mathtt{qse} > 0$,
Pq5: $q \textbf{ in } \{21 \ldots 23\} \ \wedge\ tmp.q > 0 \ \Rightarrow\ \mathtt{qse} > 0$.

Similarly, for qsm, we need the invariants:

Qq0: $\mathtt{sm} \leq \mathtt{qsm}$,
Qq1: $q \textbf{ at } 19 \ \wedge\ q \notin \mathtt{buf} \ \Rightarrow\ \mathtt{ne} > 1$,
Qq2: $q \textbf{ at } 20 \ \wedge\ \mathtt{ne} \leq 1 \ \Rightarrow\ \mathtt{qsm} > 0$,
Qq3: $q \textbf{ in } \{21 \ldots 23\} \ \wedge\ tmp.q \leq 0 \ \Rightarrow\ \mathtt{qsm} > 0$,
Qq4: $q \textbf{ in } \{25, 26\} \ \Rightarrow\ \mathtt{qsm} > 0$.

Indeed, in the proof that the concrete step 18 corresponds with the abstract step 11, we use *Pq3*, *Pq0d*, and *Mq1*. At the concrete step 26, we only need *Qq4*. At the concrete step 28, we use *Pq1d*. The other concrete steps are easily seen to correspond to skips of the abstract algorithm.

We thus have a simulation from the transition system (6) of Sect. 5.2 to system (3) of Sect. 4 that respects the processes and their doorways, and that leaves the environment actions unchanged. If we have an execution of system (6) in which some process overtakes a competing process k times, the simulation gives us an execution of system (3) in which some process overtakes a competing process k times. As system (3) satisfies bounded overtaking with bound 2, system (6) therefore also satisfies bounded overtaking with bound 2.

6. Udding's algorithm

In [Udd86], Udding, using the same shared variables as Morris, derived the algorithm (8).

At the first glance, the algorithms of Udding and Morris are rather similar. Yet, the details differ considerably. In Morris' algorithm, the plain semaphores se and sm form a split binary semaphore and the buffered semaphore sb guards the variable ne. In Udding's algorithm, the pair (sb, sm) forms a split binary semaphore, where the first one is buffered and the second one is plain. The third semaphore, se, is superfluous for mutual exclusion and for the absence of immediate deadlock, but it is necessary to ensure bounded overtaking.

Udding's algorithm can be treated in precisely the same way as Morris' algorithm. In particular, one can use more or less obvious invariants to prove mutual exclusion and absence of immediate deadlock, and one can extend it with history variables to construct a simulation to the abstract algorithm of Sect. 4.

(8)
```
process (p) =
    loop
        NCS ;
        P(sb) ; ne++ ; V(sb) ;
        P(se) ; P(sb) ; nm++ ; ne-- ;
        if ne > 0 then V(sb) else V(sm) endif ;
        V(se) ;
        P(sm) ; nm-- ; CS ;
        if nm > 0 then V(sm) else V(sb) endif
    endloop.
```

As the differences in the details are not very illuminating, we defer the complete treatment of Udding's algorithm to a technical report, see [Hes11].

7. Polite semaphores and the algorithm of Martin–Burch

In [Udd86], Udding stipulates: "For weak semaphores, one assumption is made, however, viz. a process that executes a **V**-operation on a semaphore will not be the one to perform the next **P**-operation on that semaphore, if a process has been blocked at that semaphore. One of the waiting processes is allowed to pass the semaphore."

When reading this paragraph, we first assumed that the second sentence was an essentially superfluous elaboration of the first sentence. Later, it appeared that the first sentence is superfluous because it follows from the second sentence. The second sentence suggests the buffered semaphore of Sect. 2.7. The first sentence, however, suggests a weaker kind of semaphore that only forbids the process that executed **V** most recently, to pass the semaphore.

We therefore propose the *polite* semaphore as a formalization of the first sentence of Udding's stipulation. It has two shared variables: a counter `cnt` to count the number of blocked processes, and a shared variable `last` to hold the "latest **V**-executer" while there are blocked processes. Initially $\texttt{cnt} = 0$ and $\texttt{last} = \bot$ (no process). Its operations **V** and **P** are given by

```
V(s) : ⟨ s := s + 1 ;
         if cnt > 0 then last := self endif ⟩.

P(s) : ⟨ if s > 0 ∧ self ≠ last
         then s := s − 1 ; last := ⊥ ; goto ℓ
         else cnt := cnt + 1 endif ⟩ ;
       ⟨ await s > 0 ∧ self ≠ last →
         s := s − 1 ; last := ⊥ ; cnt := cnt − 1 ⟩ ;
ℓ :    ...
```

As a reviewer noticed, the semaphore preserves the invariant $\texttt{last} = \bot \vee \texttt{cnt} > 0$. Therefore, $\mathit{self} = \texttt{last}$ implies $\texttt{cnt} > 0$: other processes are available to pass the semaphore.

As the polite semaphore only counts the number of waiting processes and does not store them, the buffered semaphore seems stronger than the polite one. A proof of this requires some formal background. We therefore postpone it to Sect. 8.2.

Section 7.1 gives scenarios to prove that algorithms of Udding and Morris are incorrect when implemented with a polite semaphore. In Sect. 7.2, we present the algorithm of Martin and Burch. A transition system version is given in Sect. 7.3, together with proofs of mutual exclusion and absence of immediate deadlock. The algorithm has an inner loop, the termination of which is fairly obvious. It is formally treated in Sect. 7.4. Section 7.5 contains a construction of a simulation to the abstract algorithm of Sect. 4.

7.1. Polite semaphores are strictly weaker than buffered ones

The algorithms of Morris and Udding do not guarantee bounded overtaking when their buffered semaphore is replaced by a polite one. We show this by means of a scenario for Udding's algorithm because the polite semaphore was inspired by Udding's phrasing.

Scenario. Consider Udding's algorithm of Sect. 6 with a polite semaphore `sb`, with its associated shared variables `last` and `cnt`. We use three processes q_0, q_1, q_2, that start in *NCS*. We show that process q_0 can remain waiting at **P**(`sb`) forever, while the processes q_1 and q_2 cycle through *CS* again and again.

The scenario consists of 9 stages:

1. q_1 passes **P**(`sb`).
2. q_0 accesses **P**(`sb`), increments `cnt`, and starts waiting.
3. q_1 passes **V**(`sb`) and becomes `last`.
4. q_2 passes **P**(`sb`) and **V**(`sb`), and becomes `last`.
5. q_1 passes **P**(`se`), **P**(`sb`), **V**(`sb`), **V**(`se`), and equals `last`.
6. q_2 passes **P**(`se`), **P**(`sb`), **V**(`sm`), and **V**(`se`).
7. q_1 passes **P**(`sm`), *CS*, **V**(`sm`), and reaches *NCS*.
8. q_2 passes **P**(`sm`), *CS*, **V**(`sb`), becomes `last`, and reaches *NCS*.
9. q_1 passes **P**(`sb`) and **V**(`sb`), and becomes `last`.

Now the state is identical to the state where stage 4 starts. The cycle can start again at stage 4, with process q_0 still waiting to pass **P**(`sb`).

For Morris' algorithm, one can use almost the same scenario with different **V** operations for q_2 in the stages 6 and 8.

7.2. The algorithm of Martin and Burch

The algorithm of Martin and Burch [MB85] implements the abstract algorithm (3) by delegating the decision `ne` $= 0$ in line 11 to the first competing process. The algorithm uses two semaphores that form a split binary semaphore. The paper supposes the axioms (2) for the buffered semaphore. We show here that it suffices to take one polite semaphore `sb` and one plain semaphore `sm`. The initial conditions are `sb` $= 1$ and `sm` $= 0$. The algorithm also uses a single shared variable `m`, initially 0, to estimate the number of competing processes.

```
process (p) =
    var n : ℤ ;
    loop
        NCS ; P(sb) ;
        if m = 0 then
            m := 1 ;  n := 0 ;
            while n ≠ m do
                n := m ; V(sb) ; P(sb)
            endwhile ;
            V(sm) ;
        else
            m++ ; V(sb) ;
        endif ;
        P(sm) ; CS ; m-- ;
        if m > 0 then V(sm) else V(sb) endif
    endloop.
```

In the first conditional statement, every process increments `m`. The first competing process takes the role of gate keeper and allows subsequent competing processes to pass to **P**(`sm`). It does so as long as there are competing processes arriving because it (or rather the semaphore `sb`) is polite. When there are no more competing processes, the gate keeper terminates its **while** loop and opens the semaphore `sm`.

(9)
```
process (p) =
    var n, tmp : ℤ ;
    loop
 9      NCS ;
10      if sb > 0 ∧ p ≠ last then sb-- ; last := ⊥ else
            cnt++ ;
11          await sb > 0 ∧ p ≠ last ; sb-- ; last := ⊥ ; cnt-- ;
        endif ;
12      if (tmp := m) = 0 then
            rex := p ;
13          m := 1 ; n := 0 ;
14          while (tmp := m) ≠ n do
15              n := tmp ;
                sb++ ; if cnt > 0 then last := p endif ;
16              if sb > 0 ∧ p ≠ last then sb-- ; last := ⊥ else
                    cnt++ ;
17                  await sb > 0 ∧ p ≠ last ; sb-- ; last := ⊥ ; cnt-- ;
                endif
            endwhile ;
18          sm++ ;
        else
19          m := tmp + 1 ;
20          sb++ ; if cnt > 0 then last := p endif ;
        endif ;
21      await sm > 0 ; sm -- ;
22      CS ;
23      tmp := m − 1 ;
24      m := tmp ;
25      if tmp ≠ 0 then sm++ else
            sb++ ; if cnt > 0 then last := p endif ;
        endif
    endloop.
```

7.3. A transition system for the Martin–Burch algorithm

The polite semaphore sb is modelled as described above with shared variables cnt and last. Just as in Sect. 5, we introduce private variables *tmp* to hold the latest value of m. We have removed superfluous inspections of m as much as possible. We introduce a shared history variable rex for the gate keeper's identity. This is not strictly necessary, but it is convenient for the proof. Note that rex can be modified in the same atomic command 12 as the inspection of m, because rex is only a history variable. In this way, we arrive at algorithm (9).

The region guarded by the split binary semaphore consists of the lines after 11, with the exceptions 16, 17, 21:

Iq0: $\#\{q \mid q \textbf{ in } \{12\ldots15, 18\ldots20, 22\ldots25\}\} + \mathtt{sb} + \mathtt{sm} = 1.$

This implies mutual exclusion because *CS* is at 22.

For the polite semaphore sb, we have three invariants concerning last and cnt:

Iq1: $\mathtt{cnt} = \#\{q \mid q \textbf{ in } \{11, 17\}\},$
Iq2: $\mathtt{last} = \bot \ \vee\ \mathtt{cnt} > 0,$
Iq3: $\mathtt{last} = q \ \wedge\ q \textbf{ in } \{11, 17\} \ \Rightarrow\ \mathtt{cnt} > 1.$

We have the obvious invariant that all processes are always in their code region:

Jq0: $q \textbf{ in } \{9\ldots25\}.$

The shared variable m counts a number of processes:

Jq1: $\mathtt{m} = \#\{q \mid q \textbf{ in } \{14 \ldots 18, 20 \ldots 24\}\}.$

As the assignments to m are guarded by the semaphores, we also have

Jq2: $\mathtt{m} = (q \textbf{ in } \{15, 19, 25\} \,?\, tmp.q$
$\quad : q \textbf{ at } 24 \,?\, tmp.q + 1$
$\quad : q \textbf{ at } 13 \,?\, 0 \; : \; \mathtt{m}).$

We turn to the question of absence of immediate deadlock. A process q is disabled if and only if:

$$(q \textbf{ in } \{11, 17\} \;\wedge\; (\mathtt{sb} = 0 \;\vee\; \mathtt{last} = q)) \;\vee\; (q \textbf{ at } 21 \wedge \mathtt{sm} = 0).$$

The first point of deadlock freedom is the invariant:

Jq3: $\mathtt{sm} > 0 \;\Rightarrow\; \exists\, q : q \textbf{ at } 21.$

Its proof is based on:

Jq4: $q \textbf{ in } \{12 \ldots 20\} \;\Rightarrow\; r \textbf{ not-in } \{22 \ldots 25\},$
Jq5: $q \textbf{ in } \{12 \ldots 20\} \;\Rightarrow\; \mathtt{sm} = 0,$
Jq6: $q \textbf{ in } \{13 \ldots 18\} \;\Rightarrow\; q = \mathtt{rex}.$

Here, we use the history variable rex introduced above.

The second point of deadlock freedom is the invariant:

Kq0: $q \textbf{ at } 21 \;\wedge\; \mathtt{sb} > 0 \;\Rightarrow\; \mathtt{rex} \textbf{ in } \{13 \ldots 18\}.$

The proof of *Kq0* relies on the additional invariants:

Kq1: $q \textbf{ in } \{19, 20\} \;\Rightarrow\; \mathtt{rex} \textbf{ in } \{16, 17\},$
Kq2: $q \textbf{ at } 12 \;\wedge\; \mathtt{m} \neq 0 \;\Rightarrow\; \mathtt{rex} \textbf{ in } \{16, 17\},$
Kq3: $\mathtt{sb} > 0 \;\wedge\; \mathtt{m} \neq 0 \;\Rightarrow\; \mathtt{rex} \textbf{ in } \{16, 17\},$
Kq4: $q \textbf{ at } 19 \;\Rightarrow\; \mathtt{m} \neq 0.$

Theorem 5 *System* (9) *has absence of immediate deadlock.*

Proof Assume that all participating processes are disabled. Then all processes are at the lines 9, 11, 17, or 21. Therefore, the invariant *Iq0* implies that $\mathtt{sb} + \mathtt{sm} = 1$. If $\mathtt{sm} > 0$, then the invariant *Jq3* implies that there is a process at line 21, which is participating and enabled. We therefore have $\mathtt{sb} > 0$. If there is a disabled process at line 21, the invariant *Kq0* implies that there is a process in the region 13–18, which process is necessarily disabled at 17.

Now assume there are participating processes. This implies that there are disabled processes at 11 or 17. One of these can be disabled by being last, but by *Iq1* and *Iq3*, there is at least one enabled process at 11 or 17. This is a contradiction. □

7.4. The inner loop terminates

In this case, the absence of immediate deadlock does not imply deadlock-freedom because system (9) has the inner loop 14–17. In order to reuse the argument used at the end of Sect. 4, we need to prove that every participating period of any process has only a bounded number of steps. This is proved as follows.

We need the assumption that the number of active processes is finite, say bounded by a number N. By *Jq1*, it then follows that $\mathtt{m} \leq N$. On the other hand, it is easy to verify the invariants:

Kq5: $q \textbf{ in } \{14 \ldots 17\} \;\Rightarrow\; n.q \leq \mathtt{m},$
Kq6: $q \textbf{ at } 15 \;\Rightarrow\; n.q < \mathtt{m}.$

We now define, for any process q, the state function

$$\begin{aligned} bf(q) = pc.q + (&pc.q \leq 13 \,?\, 0 \\ &: pc.q \leq 15 \,?\, 4 \cdot n.q \\ &: pc.q \leq 17 \,?\, 4 \cdot n.q - 4 \\ &: 4 \cdot N \;). \end{aligned}$$

Now it is clear that $bf(q)$ does not change when some process $p \neq q$ does a step. The value of $bf(q)$ increases in every step of q, except for the backward jump from 25 to 9. At line 15, this follows from *Kq6*. For the jump from 14 to 18, when $n.q = \mathtt{m}$, it follows from *Jq1*. All other cases are straightforward.

As $9 \leq bf(q) \leq 25 + 4 \cdot N$, it follows that process q does not more than $16 + 4 \cdot N$ steps in every participating period.

7.5. Refinement function

We turn to the construction of a simulation from (9) to (3). As before, the construction consists of two steps. In the first step, we introduce shared history variables `qse`, `qsm`, `qne`, and `qnm` in (9) that will map to the shared variables `se`, `sm`, `ne`, and `nm` of (3). We also introduce a state function that can serve as the abstract program counter. In the second step, we ignore the implementation variables and promote the history variables to actual variables of (3).

The crucial problem is to choose the point where the abstract semaphore `sm` is incremented. This should be the point where no more processes are waiting at 11, and the process `rex` at 16 has $n.q = \mathtt{m}$ and does not need to start waiting at 17. This choice is reflected by the guard at 16 for the modification of the history variables below, and also by the exceptional value 12 at lines 14 and 18 for the abstract program counter *apc*, below. The choice is justified at the end, when we prove that the natural projection function is a refinement function.

The history variables have the initial values $\mathtt{qse} = 1$ and $\mathtt{qsm} = \mathtt{qne} = \mathtt{qnm} = 0$. They are modified in the lines 10, 16, 20, 21 and 25, in the following way.

```
10a   qne++ ;
16a   if sb > 0 ∧ p ≠ last ∧ n = m
      then qne-- ; qnm++ ; qse-- ; qsm++ endif ;
20a   qne-- ; qnm++ ;
21a   qsm-- ; qnm-- ;
25a   if tmp ≠ 0 then qsm++ else qse++ endif.
```

In other words, the lines 10, 16, 20, 21, 25 are extended with 10a, 16a, 20a, 21a, respectively. For the history variable *qse*, we prove the invariants:

Lq0: q **in** $\{12 \ldots 20\} \;\Rightarrow\; \mathtt{qse} > 0 \;\vee\; (q$ **at** $14 \;\wedge\; n.q = \mathtt{m}) \;\vee\; q$ **at** 18,
Lq1: $\mathtt{sb} \leq \mathtt{qse}$.

For the history variable *qsm*, we prove

Lq2: $\mathtt{qsm} = \mathtt{sm} + ((\mathtt{rex}$ **at** $14 \;\wedge\; n.\mathtt{rex} = \mathtt{m}) \;\vee\; \mathtt{rex}$ **at** $18 \,?\, 1 : 0)$,
Lq3: q **at** $17 \;\wedge\; \mathtt{sb} > 0 \;\Rightarrow\; \mathtt{last} = q \;\vee\; n.q < \mathtt{m}$,
Lq4: q **in** $\{16, 17\} \;\wedge\; \mathtt{sb} = 0 \;\Rightarrow\; n.q < \mathtt{m} \;\vee\; \exists r : r$ **in** $\{12, 19\}$.

It is relatively easy to prove the invariants:

Lq5: $\#\{q \mid q$ **in** $\{11 \ldots 20\}\} = \mathtt{qne} + \mathtt{qsm} - \mathtt{sm}$,
Lq6: $\#\{q \mid q$ **at** $21\} = \mathtt{qnm} - \mathtt{qsm} + \mathtt{sm}$.

Critical properties for progress are contained in the invariants:

Lq7: q **in** $\{11 \ldots 20\} \;\wedge\; \mathtt{sb} > 0 \;\wedge\; \mathtt{rex}$ **at** $16 \;\wedge\; n.\mathtt{rex} = \mathtt{m}$
$\Rightarrow\; \mathtt{rex} = q \;\vee\; \mathtt{rex} = \mathtt{last}$,
Lq8: q **in** $\{12 \ldots 20\} \;\wedge\; \mathtt{last} = r \;\Rightarrow\; r$ **in** $\{16, 17, 21\}$.

We define the abstract program counter on the concrete state space by

$$\begin{aligned} apc.q = (&pc.q \leq 10 \,?\, pc.q \\ &: (pc.q = 14 \;\wedge\; n.q = \mathtt{m}) \;\vee\; pc.q \in \{18, 21\} \,?\, 12 \\ &: pc.q \leq 20 \,?\, 11 \\ &: pc.q = 22 \,?\, 13 \;:\; 14). \end{aligned}$$

It remains to prove that the natural projection from the extended concrete state space to the state space of (3) is a refinement function. This projection is defined by

$$f(x) = (\#\ \mathtt{se} := x.\mathtt{qse},\ \mathtt{sm} := x.\mathtt{qsm},\ \mathtt{ne} := x.\mathtt{qne},\ \mathtt{nm} := x.\mathtt{qnm},\ pc = apc\ \#).$$

In order to show that function f is well-defined, we first notice that $\mathtt{qse} \geq 0$ and $\mathtt{qsm} \geq 0$ because of the invariants *Lq1* and *Lq2*.

In order to show that f is a refinement function, we need for the lines 16 and 20 the invariants:

Lq0d: q **in** $\{16, 20\} \Rightarrow qse > 0$,
Lq7d: q **at** $16 \wedge \mathtt{sb} > 0 \wedge n.q = \mathtt{m} \Rightarrow qne = 1 \vee \mathtt{last} = q$,
Lq5d: q **at** $20 \Rightarrow qne > 1$.

Here, *Lq0d* is immediate from *Lq0*. *Lq7d* follows from *Lq7* together with *Jq6*, *Lq2* and *Lq5*. *Lq5d* follows from *Lq5* together with *Kq1* and *Lq2*.

We need *Lq3* for line 17. For line 25, we need

Lq6d: q **at** $25 \Rightarrow \mathtt{qnm} = tmp.q \geq 0$,

which follows from *Lq6*, together with *Iq0d*, *Jq1*, *Jq2*, *Jq4*, and *Lq2*.

The invariant *Iq0d* is used to ensure that, when some process is at line 14, there are no processes at 13, 19, and 24.

We thus have a simulation from the transition system (9) of Sect. 7.3 to system (3) of Sect. 4 that respects the processes and their doorways, and that leaves the environment actions unchanged. By the argument given at the end of Sect. 5.3, it follows that system (9) also satisfies bounded overtaking with bound 2.

8. Comparison between polite and buffered semaphores

As the role of `last` in the polite semaphore is not recognizable in the buffered one, it is not obvious that the buffered one simulates (implements) the polite one. The proof of this requires some formal background. Section 8.1 therefore introduces reduction of transition systems, to combine atomic steps.

8.1. Reduction

In addition to strict simulation, as introduced in Sect. 3.3, there is also a concept of general, i.e. nonstrict simulation. The idea was introduced in [AL91]. When the concrete system needs more steps than the abstract system it is supposed to simulate, the formalism allows the insertion of stutter steps in the abstract behaviour because the transition relation of the abstract system is reflexive. Nonstrict simulation is only needed in the relatively rare cases where the concrete transition system can exhibit behaviour in fewer steps than the abstract system it is supposed to simulate.

We define a *stutter function* s to be a function $\mathbb{N} \to \mathbb{N}$ that satisfies $s(0) = 0$, and $s(i+1) \in \{s(i), s(i)+1\}$ for all $i \in \mathbb{N}$, and that is unbounded. A binary relation $F \subseteq X_K \times X_L$, is called a *simulation* from K to L [Hes08] if, for every execution $(x_n \mid n \in \mathbb{N})$ of K, there is a stutter function s and an execution $(y_n \mid n \in \mathbb{N})$ of L with $(x_{s(n)}, y_n) \in F$ for all n. Note that function s makes it possible that a single state x_k of K is related to several subsequent states y_n of L. Clearly every strict simulation is a simulation (with s the identity function). Nonstrict simulation is related to the ASM-refinement of Schellhorn [Sch08].

We use nonstrict simulation in the following setting. If we have a transition system K, we may want to combine several innocent steps into a single step. This gives rise to a new transition system, the reduction of K. We formalize "innocent" by means of an equivalence relation.

Let K be a transition system and let E be an equivalence relation on the state space X_K. The *E-reduction* $\tilde{K}$ of K is obtained by allowing the steps of K within E to be accumulated in the following way. Let $M = N_K \cap E$ be the set of steps of K within E. Let M^* be the reflexive transitive closure of relation M, and let $\tilde{N}_K$ be the relational composition $M^* \circ N_K \circ M^*$. For every step $(x, y) \in \tilde{N}_K$, there are a finite sequence $(u_i \mid i \leq e)$ and a number m with $u_0 = x$ and $u_e = y$ and $(u_i, u_{i+1}) \in N_K$ for all $i < e$, and $(u_i, u_{i+1}) \in E$ for all $i < e$ with $i \neq m$. Transition system $\tilde{K}$ is defined as $\tilde{K} = (X_K, A_K, \tilde{N}_k)$.

As every step of K is a step of $\tilde{K}$, the identity relation is a simulation from K to $\tilde{K}$. On the other hand, we have

Theorem 6 *Let K be a transition system and let E be an equivalence relation on the state space X_K. Then E is a simulation from the reduction $\tilde{K}$ to K.*

Proof Let $(x_k \mid k \in \mathbb{N})$ be an execution of $\tilde{K}$. For every k, we have $(x_k, x_{k+1}) \in \tilde{N}_K$, and hence a finite sequence $(u_{k,i} \mid i \leq e(k))$ and a number $m(k)$ with $u_{k,0} = x_k$ and $u_{k,e(k)} = x_{k+1}$, and $(u_{k,i}, u_{k,i+1}) \in N_K$ for all $i < e(k)$, and $(u_{k,i}, u_{k,i+1}) \in E$ for all $i < e(k)$ with $i \neq m(k)$. We now define recursively $f : \mathbb{N} \to \mathbb{N} \times \mathbb{N}$ by

$$f(0) = (0, 0),$$
$$f(n+1) = \textbf{let } (k, r) = f(n) \textbf{ in } (r < e(k) - 1 \,?\, (k, r+1) : (k+1, 0)).$$

The sequence of states $y_n = u_{f(n)}$ is an execution of K. We define $s : \mathbb{N} \to \mathbb{N}$ by

$$s(n) = \textbf{let } (k, r) = f(n) \textbf{ in } (r \leq m(k) \,?\, k : k+1).$$

Then s is a stutter function and $(x_{s(n)}, y_n) \in E$ holds for all $n \in \mathbb{N}$. Therefore E is a simulation from $\tilde{K}$ to K. □

Remark. In this proof, the first component of f is also a stutter function, but it is not the one that we need for the simulation.

8.2. Relating buffered and polite semaphores

The buffered semaphore is stronger than the polite one. We prove this by means of refinement. For this purpose, we take a system where processes infinitely often can perform **P** and **V** operations on a semaphore `s` with the initial value s_0. The system is modelled as follows:

```
process (p) =
    loop
        choose b ∈ Boolean ;
        if b then V(s) else P(s) endif
    endloop.
```

The choice of b is performed by the environment. The system is responsible for the **V** and **P** operations, and it uses either a buffered semaphore or a polite one for this. In order to distinguish the two semaphores, we write `sb` for the buffered one and `sp` for the polite one.

The transition system *Buf* for the buffered semaphore becomes

```
Buf:    process (p) =
            loop
10              choose b ∈ Boolean ;
11              if b then (* V *)
                    if empty?(buf) then sb := sb + 1
                    else remove some q from buf  endif
                else (* P *)
                    if sb > 0 then sb := sb − 1 ;  goto 10
                    else buf := buf ∪ {p}  endif ;
12                  await p ∉ buf ;
                endif
            endloop.
```

To get a precise correspondence between the buffered system and the polite system, we model the polite version *Pol* with an additional *skip* step after waiting for the semaphore. This is necessary because, for the buffered semaphore, a process can be at 12 but no longer be blocked. In that case, it is not observably at 10, but it can go to 10 at any time. We must therefore allow the polite semaphore a similar step. The only observable aspect of our system is the set of processes for which the environment is enabled. We thus introduce the observation function *obs* given by $obs(u) = \{q \mid u.pc(q) = 10\}$ for states u.

We construct a refinement from transition system *Buf* to the reduction (see Sect. 8.1) of system *Pol* with respect to *observational equivalence*: two states u and v of system *Pol* are called observationally equivalent iff $obs(u) = obs(v)$.

Pol:
```
process (p) =
    loop
10      choose b ∈ Boolean ;
11      if b then (* V *)
            sp := sp + 1
            if cnt > 0 then last := p endif
        else (* P *)
            if sp > 0 ∧ p ≠ last
            then sp := sp − 1 ; last := ⊥ ; goto 10
            else cnt := cnt + 1 endif ;
12          await sp > 0 ∧ p ≠ last ;
            sp := sp − 1 ; last := ⊥ ; cnt := cnt − 1
13          skip
        endif
    endloop.
```

Because a process $q \notin$ `buf` at line 12 of *Buf* can go to line 10, just as if it were at line 13 of *Pol*, we define function *apc* for state x of *Buf* and process q by

$$apc(x)(q) = (x.pc.q = 12 \ \wedge \ q \notin x.\texttt{buf} \ ? \ 13 : x.pc.q).$$

Similarities between the actions on `buf` in *Buf* and the actions of `cnt` in *Pol* suggest that `cnt` should represent the number of elements of `buf`. In this way, after some experimentation, we arrive at the function f from the state space of *Buf* to the state space of *Pol* given by

$$\begin{aligned} f(x) = (\#\ &\texttt{sp} := x.\texttt{sb} \ , \ \texttt{cnt} := \#(x.\texttt{buf}) \ , \\ &\texttt{last} := \bot \ , \ pc := apc(x) \ , \ b := x.b \quad \#). \end{aligned}$$

Now step 10 of *Buf* corresponds to step 10 of *Pol*. Step 12 of *Buf* corresponds to step 13 of *Pol*. Step **V** of *Buf* with empty `buf` corresponds to step **V** of *Pol*. Step **P** of *Buf* corresponds to step **P** of *Pol* because of the obvious invariant of *Buf* that $q \in$ `buf` implies that q is at 12.

In the definition of function f, we ignore variable `last`. We compensate for this by extending system *Pol* with a sequential composition of the step that sets `last` with a step that resets it. Indeed, the step **V** of *Buf* that removes some q from a nonempty `buf` corresponds to step **V** of *Pol* followed by step 12 of *Pol* for process q. We therefore add to transition system *Pol* the sequential composition in which the processes p and q do these steps immediately one after the other:

$$\begin{aligned} &stepVP(p, q) = \\ &\quad pc.p = 11 \ \wedge \ b.p \ \wedge \ pc.q = 12 \ \wedge \ \texttt{cnt} > 0 \quad \rightarrow \\ &\qquad pc.p := 10 \ ; \ pc.q := 13 \ ; \ \texttt{cnt--} \ ; \ \texttt{last} := \bot. \end{aligned}$$

Here the notation $B \rightarrow S$ means that the transition can only be taken when B holds, and then has the effect of S.

Adding the steps $stepVP(p, q)$ gives an extended transition system, more nondeterministic, but it has no new reachable states. Function f is a refinement function from the buffered system *Buf* to this extended system. More generally, as step 12 of *Pol* has no effect on the observation function *obs*, function f is a refinement function from system *Buf* to the reduction of this extension of *Pol* for observational equivalence.

It follows that the graph of f is a simulation from system *Buf* to this reduction. By Theorem 6, observational equivalence is a simulation from this reduction to system *Pol*. The composition of these simulations is a simulation from the buffered semaphore *Buf* to the polite one *Pol*.

The above proof still leaves the possibility that the buffered semaphore when implementing the polite one, would introduce additional deadlock. This is not the case. Assume that the system *Buf* is in a state with immediate deadlock. This means that there are participating processes (i.e. processes not at 10), and they are at 12 and contained in `buf`. System *Buf* has the easy invariant that `sb` > 0 implies that `buf` is empty. It therefore follows that `sb` $= 0$. The corresponding state of *Pol* therefore has `sp` $= 0$ with the same processes at 12, and hence is also in immediate deadlock.

Remark. We have to consider additional deadlock separately, because the transition systems used here are simplifications of the specifications of [AL91] and have no liveness properties. If we had used the full formalism of [AL91], this could have blown up the paper by some 10 %. The only benefit would have been, that the paragraph just before this remark would not have come as an afterthought but would have to be integrated in the preceding proof. For the rest of the proofs, it would have added some trivial verifications. The PVS proof could have grown by some 20 % because of the richer formalism.

9. Dijkstra's conjecture

Dijkstra's conjecture that a strong semaphore cannot be implemented by weak semaphores, was refuted in [Mor79, MB85, Udd86] using a buffered semaphore. For refutation [MB85], a polite semaphore is sufficient. We therefore propose to rephrase the conjecture as follows. *It is not possible to implement starvation-free mutual exclusion for an unbounded number of processes with ordinary variables and plain semaphores, under weak fairness, such that the system satisfies the principle of single critical reference.*

We have to be careful, however. Firstly about the principle of critical reference. The conjecture fails when we have an atomic counter, i.e., a shared integer variable that can atomically be incremented, decremented, and tested for equality with 0. This is shown by algorithm (4).

The unboundedness of the set of processes is needed, and the memory model must be carefully constrained, because there are mutual exclusion algorithms, even with FCFS, that need no semaphores or other locks at all [Lam74, LH91, Szy90]. These work for boundedly many processes, and even for unboundedly many processes when concurrently expandable arrays are available. On the other hand, plain semaphores or mutexes are enough when we replace weak fairness by strong fairness. Therefore, if one wants to prove the conjecture, one needs a careful restriction of the available programming repertoire.

10. Conclusion

In this paper, refinement serves two purposes. On the one hand, it is used to show that each of the four algorithms that refine the abstract algorithm have bounded overtaking. On the other hand, it formalizes the similarity between these algorithms. It seems likely that the abstract algorithm is also refined by Szymanski's algorithm [Szy90, Figure 1], which uses 3 shared bits per process and no semaphores.

Refinement is not useful in the proofs of mutual exclusion, because we need the mutual exclusion invariants in the proof of the refinement. The refinement is also useless for the proofs of absence of immediate deadlock, by the nature of refinement and deadlock.

In our view, our definitions of the buffered and polite semaphores in terms of transition systems are much clearer than the axiomatic approach of [MB85, MvdS89]. The invention of the polite semaphore was inspired by Udding's vagueness about his semaphores.

Despite the popularity of lockfree algorithms and transactional memory in concurrency research, we expect that semaphores and other locking mechanisms will be used in the future as they were in the past. The issue of starvation deserves to be treated carefully. It could be an interesting research question to construct a starvation free mutual exclusion algorithm by means of mutexes and condition variables.

References

[AdBO09] Apt KR, de Boer FS, Olderog E-R (2009) Verification of sequential and concurrent programs. Springer, New York
[AL91] Abadi M, Lamport L (1991) The existence of refinement mappings. Theor Comput Sci 82:253–284
[And00] Andrews GR (2000) Foundations of multithreaded, parallel, and distributed programming. Addison Wesley, Reading
[But97] Butenhof DR (1997) Programming with POSIX threads. Addison-Wesley
[CM88] Chandy KM, Misra J (1988) Parallel program design: a foundation. Addison-Wesley
[Dij65] Dijkstra EW (1965) Solution of a problem in concurrent programming control. Commun ACM 8:569
[Dij68a] Dijkstra EW (1968) Co-operating sequential processes. In: Genuys F (ed) Programming languages. NATO Advanced Study Institute. Academic Press, London, pp 43–112
[Dij68b] Dijkstra EW (1968) The structure of the THE multiprogramming system. Commun ACM 11:341–346

[Dij77] Dijkstra EW (1977) A strong P/V-implementation of conditional critical regions. Tech rept, Tech Univ Eindhoven, EWD 651. www.cs.utexas.edu/users/EWD
[Fra86] Francez N (1986) Fairness. Springer
[Hes06] Hesselink WH (2006) Splitting forward simulations to cope with liveness. Acta Inf 42:583–602
[Hes08] Hesselink WH (2008) Universal extensions to simulate specifications. Inf Comput 206:108–128
[Hes11] Hesselink WH (2011) Starvation-free mutual exclusion with semaphores. http://www.cs.rug.nl/~wim/mechver/fairMXsema.html
[HHS86] He J, Hoare CAR, Sanders JW (1986) Data refinement refined. In: Robinet B, Wilhelm R (eds) ESOP 86. LNCS, vol 213. Springer, New York, pp 187–196
[Hoa74] Hoare CAR (1974) Monitors: an operating system structuring concept. Commun ACM 17:549–557
[HS08] Herlihy M, Shavit N (2008) The art of multiprocessor programming. Morgan Kaufmann
[Lam74] Lamport L (1974) A new solution of Dijkstra's concurrent programming problem. Commun ACM 17:453–455
[Lea00] Lea D (2000) Concurrent programming in Java. Addison-Wesley
[LH91] Lycklama EA, Hadzilacos V (1991) A first-come-first-served mutual-exclusion algorithm with small communication variables. ACM Trans Program Lang Syst 13:558–576
[LPS81] Lehmann D, Pnueli A, Stavi J (1981) Impartiality, justice and fairness: the ethics of concurrent termination. In: Proc 8th ICALP. LNCS, vol 115. Springer, Berlin, pp 264–277
[LV95] Lynch N, Vaandrager F (1995) Forward and backward simulations. Part I: untimed systems. Inf Comput 121:214–233
[MB85] Martin AJ, Burch JR (1985) Fair mutual exclusion with unfair P and V operations. Inf Process Lett 21:97–100
[Mil71] Milner R (1971) An algebraic definition of simulation between programs. In: Proc 2nd int joint conf on artificial intelligence. British Comp Soc, pp 481–489
[Mor79] Morris JM (1979) A starvation-free solution to the mutual exclusion problem. Inf Process Lett 8:76–80
[MvdS89] Martin AJ, van de Snepscheut JLA (1989) Design of synchronization algorithms. In: Broy M (ed) Constructive methods in computing science. Springer, Berlin, pp 445–478
[OG76] Owicki S, Gries D (1976) An axiomatic proof technique for parallel programs. Acta Inf 6:319–340
[OSRSC01] Owre S, Shankar N, Rushby JM, Stringer-Calvert DWJ (2001) PVS version 2.4, system guide, prover guide, PVS language reference. http://pvs.csl.sri.com
[Sch08] Schellhorn G (2008) Completeness of ASM refinement. Electron Notes Theor Comput Sci 214:25–49
[Szy90] Szymanski BK (1990) Mutual exclusion revisited. In: Proceedings of the fifth Jerusalem conference on information technology. IEEE Computer Society, pp 110–117
[Tan08] Tanenbaum AS (2008) Modern operating systems, 3rd edn. Pearson Education/Prentice Hall
[Udd86] Udding JT (1986) Absence of individual starvation using weak semaphores. Inf Process Lett 23:159–162

Received 25 May 2011
Revised 20 September 2011
Accepted 17 November 2011 by Eerke Boiten
Published online 16 December 2011